J. Parallel Distrib. Comput. 106 (2017) 14–30

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A parallel Quantized State System Solver for ODEs

Joaquín Fernandez *, Ernesto Kofman, Federico Bergero

Laboratorio de Sistemas Dinámicos, CIFASIS-CONICET, Rosario, Argentina

HIGHLIGHTS

- Novel parallelization techniques for Quantized State System (QSS) ODE simulation are proposed.
- A theoretical analysis of the parallelization error introduced is presented.
- The novel techniques are implemented on multi-core architecture.
- The presented implementation is deeply evaluated on four large scale models.

ARTICLE INFO

Article history:
Received 27 July 2016
Received in revised form
16 February 2017
Accepted 22 February 2017
Available online 7 March 2017

Keywords:
Parallel ODE simulation
QSS
Hybrid systems
Discrete event systems

ABSTRACT

This work introduces novel parallelization techniques for Quantized State System (QSS) simulation of continuous time and hybrid systems and their implementation on a multi-core architecture. Exploiting the asynchronous nature of QSS algorithms, the novel methodologies are based on the use of non-strict synchronization between logical processes. The fact that the synchronization is not strict allows to achieve large speedups at the cost of introducing additional numerical errors that, under certain assumptions, are bounded depending on some given parameters.

Besides introducing the parallelization techniques, the article describes their implementation on a software tool and it presents a theoretical analysis of the aforementioned additional numerical error. Finally, the performance of the novel methodology and its implementation is deeply evaluated on four large scale models.

1. Introduction

The modeling and simulation (M&S) of continuous systems is at the core of scientific research and different engineering areas. Every year, M&S practitioners develop larger and more complex models and their simulations carry high computational costs. With the advent of multi-core processors and multi-node clusters of computers, the parallel simulation of continuous time systems became the usual way to reduce execution times of these simulations.

Continuous time models are usually expressed as (or transformed to) sets of Ordinary Differential Equations (ODE), where numerical integration algorithms must be applied in order to solve them. Most of these algorithms are based on time-discretization [8,19], i.e. they compute the value of all state variables at some given time points.

There are certain problems in which the usage of these classic numerical algorithms yields huge computational costs. In particular, in very large systems (millions of state variables), the evaluations of the model functions may require millions of computations. In addition, if the system is stiff (i.e, in presence of fast and slow dynamics), implicit algorithms must be used that call for huge matrix inversions on several simulation steps. These problems are worsened in ODEs having frequent discontinuities, where the algorithms must detect their occurrence and restart after each of them.

Over the years, in order to mitigate the huge computational costs associated to these types of problems, various approaches for the parallel implementation of the numerical algorithms have been proposed [40,29–31].

There is a newly developed family of ODE numerical integration methods called QSS that replace the time discretization of the classic algorithms by the quantization of the state variables [28,8]. QSS integration methods have certain features (sparsity exploitation, efficient discontinuities handling [25], explicit stiff integration [33]) that reduce the computational costs in the simulation of large scale systems [17,39,4]. These facts motivated the usage of QSS methods in several applications, including Building and Power

* Corresponding author.
E-mail addresses: fernandez@cifasis-conicet.gov.ar (J. Fernandez), kofman@cifasis-conicet.gov.ar (E. Kofman), bergero@cifasis-conicet.gov.ar (F. Bergero).

http://dx.doi.org/10.1016/j.jpdc.2017.02.011
0743-7315/

Systems simulation [44,9] (where there are also plans to include these algorithms in the EnergyPlus software package [47]), as well as large biological models [2], water distribution models [38], wildfire propagation [46], and simulation of high energy particles [43] (where there is also a preliminary implementation of QSS algorithms in the software Geant4).

QSS methods are asynchronous in the sense that each state variable evolves at a different pace. Thus, if a loosely coupled large scale model is split into two or more parts, they only need to interact at the steps corresponding to changes in variables that are common to different sub-models. This fact enables to spread the simulation over different computing units so that each one integrates a different sub-model.

Based on this idea, a parallel implementation of QSS methods for a multi-core architecture was presented in [6]. That work used a discrete event implementation of the QSS algorithms on a software called PowerDEVS [5], and the synchronization between different sub-models was achieved using a real-time clock. A limitation of that approach was that discrete event implementations of QSS are inefficient [11] and the technique required the usage of a real time operating system, what restricted their usage on general purpose computers.

In this work, we present two novel techniques for parallel simulation with QSS algorithms that do not require the usage of real-time OS. Here, the synchronization between sub-models is non-strict, allowing a bounded difference between the local simulation times. We formally show that this bounded de-synchronization only introduces a bounded numerical error additional to that introduced by the QSS approximation.

We also describe the implementation of these techniques on the Stand Alone Quantized State System Solver [11], a tool implementing the whole family of QSS algorithms that is more than one order of magnitude faster than PowerDEVS. This implementation, based on native POSIX threads, targets shared memory architectures.

In order to evaluate the performance of the implementation, we perform a deep analysis on four large scale problems where it had been shown that the sequential implementation of QSS algorithms is very efficient.

The article is organized as follows. Section 2 introduces the main concepts used along the article and reviews the state of the art and related work in the area of parallel ODE simulation. Then, Section 3 presents a new parallelization technique and its implementation on the QSS Stand Alone Solver. Later, Section 4 analyzes the numerical error introduced by this technique and, based on this analysis, Section 5 proposes a second synchronization technique that performs an adaptive adjustment of the synchronization parameter. Section 6 presents the application of these two techniques to four large scale models, and Section 7 concludes the article and discusses about future work.

2. Background

2.1. Quantized state system methods

Quantized State System (QSS) methods replace the time discretization of classic numerical integration algorithms by the quantization of the state variables.

Given the ODE

$$\dot{\mathbf{x}}_a(t) = \mathbf{f}(\mathbf{x}_a(t), t) \tag{1}$$

with $\mathbf{x}_a \in \mathbb{R}^n$, the first order Quantized State System method (QSS1) [28] approximates it by

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), t). \tag{2}$$

Here, $\mathbf{x}$ is the state vector (of the numerical solution), and $\mathbf{q}$ is the *quantized state vector*. Each entry $q_j(t)$, with $j = 1, \ldots, n$, is related to that of the state vector $x_j(t)$ by the following *hysteretic quantization function*:

$$q_j(t) = \begin{cases} x_j(t) & \text{if } |x_j(t) - q_j(t^-)| \geq \Delta Q_j \\ q_j(t^-) & \text{otherwise} \end{cases} \tag{3}$$

where ΔQ_j is called *quantum* and $q_j(t^-)$ denotes the left-sided limit of $q_j(\cdot)$ at time t.

It can be easily seen that $q_j(t)$ follows a piecewise constant trajectory that only changes when the difference between $q_j(t)$ and $x_j(t)$ becomes equal to the quantum. After each change in the quantized variable, it results that $q_j(t) = x_j(t)$.

Due to the particular form of the trajectories, the solution of Eq. (2) is straightforward and can be easily translated into a simple simulation algorithm.

For $j = 1, \ldots, n$, let t_j denote the next time at which $\left|q_j(t) - x_j(t)\right| = \Delta Q_j$. Then, the QSS1 simulation algorithm works as follows:

Algorithm 1: QSS1.

```
1  while(t < t_f) // simulate until final time tf
2    t = min(t_j), j ∈ [1, n] // adavance simulation time
3    i = argmin(t_j), j ∈ [1, n] // the i-th quantized
         state changes first
4    e_xi = t − tx_i // elapsed time since last xi update
5    x_i = x_i + ẋ_i · e_xi // update i-th state value
6    q_i = x_i // update i-th quantized state
7    t_i = min(τ > t) subject to |q_i − x_i(τ)| = ΔQ_i //
         compute next i-th quantized state change.
          Here x_i(τ) = x_i + ẋ_i · (τ − t)
8    for each j ∈ [1, n] such that ẋ_j depends on q_i
9      e_xj = t − tx_j // elapsed time since last xj
           update
10     x_j = x_j + ẋ_j · e_xj // update j-th state value
11     tx_j = t // last xj update
12     ẋ_j = f_j(q, t) // recompute j-th state derivative
13     t_j = min(τ > t) subject to |q_j − x_j(τ)| = ΔQ_j //
           recompute j-th quantized state changing
            time. Here x_j(τ) = x_j + ẋ_j · (τ − t)
14   end for
15   tx_i = t // last xi update
16 end while
```

Notice that line 8 requires that the algorithm knows which state derivatives depend on each state variable, i.e., the implementation of a QSS1 solver needs structural information about the model.

The QSS1 method has the following features:

- The quantized states $q_j(t)$ follow piecewise constant trajectories, and the state variables $x_j(t)$ follow piecewise linear trajectories.
- The state and quantized variables never differ more than the quantum ΔQ_j. This fact ensures stability and global error bound properties [28].
- The quantum ΔQ_j of each state variable can be chosen to be proportional to the state magnitude, leading to an intrinsic relative error control [27].
- Each step is local to a state variable x_j (the one which reaches the quantum change), and it only provokes evaluations of the state derivatives that explicitly depend on it.
- The fact that the state variables follow piecewise linear trajectories makes very easy to detect discontinuities. Moreover, after a discontinuity is detected, its effects are not different to those of a normal step. Thus, QSS1 is very efficient to simulate discontinuous systems [25].

However, QSS1 has some limitations as it only performs a first order approximation, and it is not suitable to simulate stiff systems.

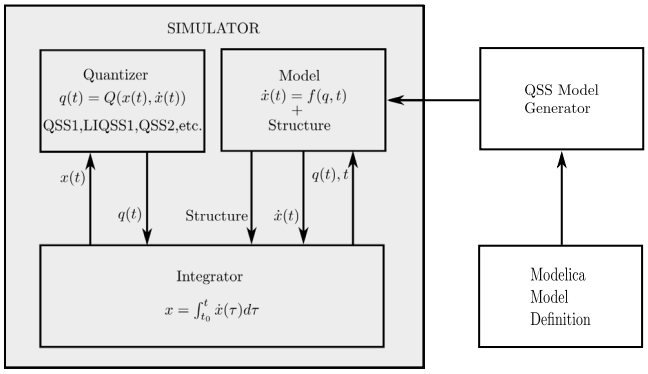

Fig. 1. Stand alone QSS solver — basic interaction scheme.

The first limitation was solved with the introduction of higher order QSS methods like the second order accurate QSS2 [24], where the quantized state follows piecewise linear trajectories, and the third order accurates QSS3 [26], where the quantized state follows piecewise parabolic trajectories.

Regarding stiff systems, a family of Linearly Implicit QSS (LIQSS) methods of order 1 to 3 was proposed in [33]. LIQSS methods are explicit algorithms that can efficiently integrate stiff systems provided that they have certain structure.

All QSS and LIQSS methods share the representation of Eq. (2). They only differ in the way that q_i is computed from x_i. Thus, their simulation algorithms are very similar to that of Algorithm 1.

2.2. The stand-alone QSS solver

As mentioned above, QSS methods replace time discretization of classic integration algorithms by the quantization of the state variables leading to a discrete event approximation of the original model. As a consequence, most QSS implementations are based on DEVS (Discrete EVent System Specification) [5]. Although these implementation are simple, the overhead imposed by the synchronization and event transmission mechanism of the DEVS simulation engine makes them inefficient. The stand-alone QSS solver presented in [11] implements the entire family of QSS methods in an efficient way without using a DEVS simulation engine improving simulation times in more than one order of magnitude.

The stand-alone QSS solver simulates models that can contain discontinuities. These models are represented as follows:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, \mathbf{d}, t) \tag{4}$$

where $\mathbf{d}$ is a vector of discrete variables that can only change when a condition

$$ZC_i(\mathbf{x}, \mathbf{d}, t) = 0 \tag{5}$$

for some $i \in \{1, \ldots, z\}$ is met. The components ZC_i form a vector of zero-crossing functions $\mathbf{ZC}(\mathbf{x}, \mathbf{d}, t)$. When a zero-crossing condition of Eq. (5) is verified, the state and discrete variables can change according to the corresponding event handler:

$$(\mathbf{x}(t), \mathbf{d}(t)) = H_i(\mathbf{x}(t^-), \mathbf{d}(t^-), t) \tag{6}$$

These models are simulated using QSS methods that approximate Eq. (4) by

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}, \mathbf{d}, t) \tag{7}$$

where each component $q_i(t)$ is a piecewise polynomial approximation of the corresponding component of the state $x_i(t)$.

The simulation is performed by three modules interacting at runtime:

1. The *Integrator*, that integrates Eq. (7) assuming that the piecewise polynomial quantized state trajectory $\mathbf{q}(t)$ is known.
2. The *Quantizer*, that computes $\mathbf{q}(t)$ from $\mathbf{x}(t)$ according to the QSS method in use and their tolerance settings (there is a different *Quantizer* for each QSS method). That way, it provides the polynomial coefficients of each quantized state $q_i(t)$ and computes the next time at which a new polynomial section starts (i.e., when the condition $|q_i(t) - x_i(t)| = \Delta Q_i$ is met).
3. The *Model*, that computes the scalar state derivatives $\dot{x}_i = f_i(\mathbf{q}, \mathbf{d}, t)$, the zero-crossing functions $ZC_i(\mathbf{x}, \mathbf{d}, t)$, and the corresponding event handlers $H_i(\mathbf{q}, \mathbf{d}, t)$. Besides, it provides the structural information required by the algorithms.

The structure information of the *Model* is automatically extracted at compile time by a *Model Generator* module. This module takes a standard model described in a subset of the Modelica language [13] and produces an instance of the *Model* module as required by the QSS solver including the structure information and the possibility of separately evaluating scalar state derivatives.

Fig. 1 shows the basic interaction scheme between the three modules mentioned above.

The structural information is comprised in four binary incidence matrices:

- *SD* (states to derivatives) is such that $SD_{i,j} = 1$ indicates that x_i is involved in the calculation of $\dot{x}_j$.
- *SZ* (states to zero-crossing functions) is such that $SZ_{i,j} = 1$ indicates that x_i is involved in the calculation of ZC_j.
- *HD* (handlers to state derivatives) is such that $HD_{i,j} = 1$ indicates that the execution of handler H_i changes some state or discrete variable involved in the calculation of $\dot{x}_j$.
- *HZ* (handlers to zero-crossing functions) is such that $HZ_{i,j} = 1$ indicates that the execution of handler H_i changes some state or discrete variable involved in the calculation of ZC_j.

Taking into account that most large systems are loosely coupled, the structure matrices are stored in a sparse form.

The simulation is carried on by the *Integrator* module, that advances the simulation time executing the simulation steps. Each simulation step may correspond to a change in a quantized variable q_i or to the execution of an event handler H_i triggered by a zero-crossing condition $ZC_i(t) = 0$. The integrator stores the state, quantized state and discrete state values x_i, q_i, and d_i, respectively. It also stores the time of the next change in each quantized state tx_i and the time of the next crossing of each zero-crossing function tz_i.

The main simulation routine at the *Integrator* module looks as follows:

Algorithm 2: QSS Integrator Module

```
1  while t < tf //while simulation time t is less
       than the final time tf
2    tx = min(txj) // time of the next change in a
       quantized variable
3    tz = min(tzj) // time of the next zero-
       crossing time
4    t=min(tx, tz) //advance simulation time
5    if t = tx then //quantized state change
6      i = argmin(txj) // the i-th quantized state
         changes first
7      Quantized_State_Step(i) //execute a
         quantized state change on variable i
8    else //zero--crossing
9      i = argmin(tzj) // the i-th event handler
         is executed first
10     Event_Handler_Step(i) //execute the
         procedure for the i-th event handler
11   end if
12 end while
```

When the next step corresponds to a change in a quantized variable q_i at time t, the *Integrator* proceeds as follows:

Algorithm 3: QSS Integrator Module — Quantized State Change

```
Quantized_State_Step(i)
{
  integrateState(x_i, ẋ_i, t) // integrate i-th
      state up to time t.
  Quantizer.update(x_i, q_i) // update i-th
      quantized state q_i
  tx_i = Quantizer.nextTime(x_i, q_i) //compute next
      i-th quantized state change time
  for each j such that SD_{i,j} = 1
    integrateState(x_j, ẋ_j, t) // integrate j-th
        state up to time t.
    ẋ_j = Model.f_j(q(t), d(t), t) // recompute j-th state
        derivative
    tx_j = Quantizer.nextTime(x_j(t), q_j(t)) //
        recompute next j-th quantized state
        change time
  end for
  for each j such that SZ_{i,j} = 1
    zc_j = Model.zc_j(q, d(t), t) // recompute j-th zero
        --crossing function
    tz_j = nextEventTime(zc_j) //recompute next j-th
        zero--crossing time
  end for
}
```

Similarly, when the next step corresponds to the execution of an event handler H_i at time t, the *Integrator* proceeds as follows:

Algorithm 4: QSS Integrator Module — Event Handler Execution

```
Event_Handler_Step(i)
{
  Model.H_i(q(t), d(t), t) //execute i-th event handler
  for each j such that HD_{i,j} = 1
    integrateState(x_j, ẋ_j, t) // integrate j-th
        state up to time t.
    ẋ_j = Model.f_j(q(t), d(t), t) // recompute j-th state
        derivative
    tx_j = Quantizer.nextTime(x_j(t), q_j(t)) //
        recompute next j-th quantized state
        change time
  end for
  for each j such that HZ_{i,j} = 1
    zc_j = Model.zc_j(q, d(t), t) // recompute j-th zero
        --crossing function
    tz_j = nextEventTime(zc_j) //recompute next j-th
        zero--crossing time
  end for
}
```

2.3. Parallel simulation

Simulations often require performing several expensive calculations. In order to obtain faster results, the usage of parallel computations is a common solution. As the current work deals with parallel simulation of ODEs using discrete event approximations, we shall mention some existing solutions in both fields: parallel simulation of ODEs and discrete event systems. Also, existing approaches for the parallel simulation with QSS methods are described as the most related results to our methodology.

2.3.1. Parallel ODE simulation

Different strategies have been proposed over the years for parallel ODE simulation based on classic numerical integration algorithms. These strategies are characterized according to the computations that they parallelize. The main categories are as follows:

- *parallelism across the model*: This technique is based on partitioning the mathematical model in a way that different components of function **f** in Eq. (1) are computed in parallel.
- *parallelism across the method*: In this approach, parallelism is achieved by executing the intrinsic calculations of the numerical method in parallel. In most implementations, this approach is combined with the parallelization across the model.
- *parallelism across the time steps*: Here, the successive integration steps of the simulation are computed in parallel.

A comparative analysis between the two first approaches is presented in [34].

An implementation that exploits efficient memory access using Iterated Runge–Kutta methods in shared memory architectures can be found in [29,30,23]. In these works, parallelizing simultaneously across the methods, models and time steps, speedups of up to 300 times are achieved using 480 cores of a supercomputer in the simulation of ODEs with up to 8,000,000 state variables.

In [18], different integration methods are benchmarked on both CPU and GPU platforms. Similarly, in [36], two Runge–Kutta methods are tested on CPU and GPU achieving speed-ups up to 115x for the GPU case.

Different applications of the parallelism across steps approach for classic numerical methods are presented and studied in [31,10,1,16,20].

2.3.2. Parallel simulation of discrete event systems

Parallelization of Discrete Event System simulations has been studied for some years now [14,15]. The basic idea is to split the model into several sub-models and to simulate them concurrently on different *logical processors* (LPs), each one having its own logical time. As the sub-models are usually inter-dependent, the logical time of the different sub-simulations must be synchronized in order to satisfy the *causality constraint*, i.e., the preservation of the chronological order of the events.

The literature divides the different approaches into three categories:

- *Conservative* algorithms, like CMB [35], where there is a synchronization mechanism that forces each LP to wait until it is sure that it will not receive messages from "the past". Conservative approaches usually achieve small speedups and suffer from certain issues (like possible deadlocks). They can be enhanced with the usage of *LookAhead* strategies [21].
- *Optimistic* algorithms, where each LP advances its logical time as much as it can and it rolls-back when an inconsistency among the interchanged messages is found. Optimistic approaches allow more parallel computations than conservative techniques at the expense of having to save intermediate simulation states and costly roll-back mechanisms. Algorithms like TimeWarp [22,41] are examples of the optimistic approach. A recent work [3] shows results for TimeWarp running a MPI based parallel simulation on almost two million cores obtaining super-linear speed-ups.
- Finally, in [42], the idea of completely avoiding the synchronization is studied, i.e. each LP advances its logical time as fast as it can disregarding of what the other LP logical time is. The results show that this approach achieves larger speedups than the previous ones at the cost of introducing errors due to violation of the causality constraint.

2.3.3. Parallel simulation with QSS methods

The asynchronous nature of QSS methods simplifies the parallelization of their computations. As it was mentioned in

Section 2.2, QSS methods can be represented as discrete events system, thus the corresponding parallelization techniques of Section 2.3.2 could be used, in principle.

However, neither of these approaches fits well in the context of QSS algorithms. In this case, strict synchronization does not allow almost any concurrent computations as it was shown in [6]. Also, optimistic methods would require huge amounts of memory to implement the roll-back mechanisms on large systems. Finally, totally unsynchronized techniques would introduce unacceptable numerical errors.

An initial study was done in [37] applying optimistic algorithm to parallelize QSS over a distributed-memory cluster showing a two-fold speedup simulating on four processors. Then, in [32], the authors present an implementation of QSS methods over shared-memory GPU architectures (Nvidia Tesla C1060, and Nvidia GeForce 8600). Preliminary results report speedups of up to eight times for models with 64 state variables. The results were not extended to larger models due to limitations on the GPU architecture (diverging branches, lack of synchronization, etc.).

Another related work was presented in [5] where two parallelization techniques, SRTS and ASRTS, are introduced for a shared-memory multi-core architecture. The synchronization amongst the different sub-simulations in this case is achieved using a real-time clock. Results show an almost linear scaling of speed-up with respect to the number of logical processors achieving speed-ups of up to 9 times for 12 logical processors. The cited work also analyzes the error introduced by these techniques showing that for a bounded difference between the logical time of each sub-simulation a bounded numerical error is introduced. This implementation was based on the PowerDEVS tool [5].

In conclusion, previous to the current work, the parallelization of QSS methods was limited to relatively small problems (a few thousands of state variables, at most) and implemented over different architectures using small numbers of processors.

3. A novel parallel QSS simulation methodology

In this section, we describe a novel technique for parallel simulation with QSS algorithms and its implementation in the Stand-Alone QSS Solver. We first present the basic idea, and then we introduce the algorithms and implementation issues corresponding to the different components of the parallel solver.

3.1. Basic idea

The presented parallelization technique is based on partitioning the model into P sub-models, so that each sub-model is simulated by a different logical process.

At each simulation step of every sub-model, the corresponding LP checks if the variable that changed must be communicated to other LPs using structural information. If so, the new values and the corresponding time-stamp are informed through an inter-process communication mechanism.

In order to avoid large errors introduced by new values of a quantized variable arriving at wrong time instants, the technique enforces the different processes to have a bounded difference between their logical simulation times. This difference is bounded by a user defined parameter Δt.

This non-strict synchronization is achieved by computing a global virtual time gvt (equal to the minimum logical time of all LPs) and not allowing the LPs to advance beyond $gvt + \Delta t$. For that goal, whenever an LP schedules its next simulation step beyond $gvt + \Delta t$, it enters a waiting routine until gvt advances or a change in a variable computed by another LP is detected.

3.2. Basic structure of the parallel solver

The parallel extension of the QSS solver targets a MIMD (multiple instruction multiple data) multi-core architecture. It uses a partitioned model of Eqs. (4)–(6), where each LP simulates a different sub-model. Every LP is composed of an *Integrator* module, a *Quantizer* module, and a *Model* module, like in the sequential simulation described in Section 2.2.

Taking into account that the state derivatives computed in a LP may depend on the states computed in another LP, it is clear that the different LPs must communicate during the simulation. Furthermore, the computation of a state derivative $\dot{x}_i$ at time t requires the knowledge of the quantized states involved in the calculation of $f_i(\mathbf{q}, \mathbf{d}, t)$ at the same time t, thus, a synchronization mechanism between the different LPs is also necessary.

Towards this goal, the sequential *Integrator* module is modified with the addition of communication and synchronization mechanisms. The basic scheme of the parallel QSS solver is shown in Fig. 2.

Next, after discussing the model partitioning issues, we describe in detail the communication and synchronization mechanisms and the resulting parallel simulation algorithm and their implementation.

3.3. Model partition

The QSS parallel simulation algorithm requires that the original model is split into p sub-models, so that each sub-model is simulated by a different LP.

In order to obtain an effective parallelization, the model partitioning should be balanced, and, at the same time, the communication between different LPs must be minimized. In some simple cases, when the system has a regular structure, this partitioning can be manually done.

In general cases, algorithms for automatic partitioning must be applied. A set of graph-theoretic sub-optimal partitioning algorithms for QSS parallel simulation was developed in [12].

In the context of this work, we shall assume that a *suitable* partition is provided.

The partitioning shall be represented by two arrays PX, and PZ, with their entries taking values in the set $\{1, \ldots, p\}$. An entry $PX_i = k$ says that the state x_i is computed by the kth LP. Similarly, an entry $PZ_i = k$ says that the zero-crossing function ZC_i and its corresponding event handler H_i are evaluated and executed by the kth LP.

In the actual implementation of the parallel QSS solver, the end user can choose between the different automatic partitioning algorithms described in [12] or he can provide a manually generated partition.

3.4. Inter-process structure

As we explained in Section 2.2, the structure information of the model is comprised in matrices SD, SZ, HD, and HZ. These structural matrices represent the direct influences of states and event handlers in state derivatives and zero-crossing functions.

Once the model is partitioned into p sub-models (according to arrays PX and PZ), it can happen that a change in a state variable – or the execution of an event handler – computed at the kth sub-model has a direct influence on some state derivatives or zero-crossing functions computed at different sub-models.

Thus, an inter-process communication mechanism is necessary, which in turn requires the knowledge of inter-process structure information.

In our implementation, this information is provided by two incidence matrices at each LP:

- SO^k is such that $SO^k_{i,l} = 1$ indicates that the ith state of the kth sub-model influences on some state derivative or zero-crossing function computed by the lth sub-model.
- HO^k is such that $HO^k_{i,l} = 1$ indicates that the variables changed by the ith event handler execution at the kth sub-model influences on some state derivative or zero-crossing function computed by the lth sub-model.

Matrices SO^k and HO^k are computed at initialization time based on the structure matrices SD, SZ, HD, HZ, and the partition arrays PX and PZ. Like the remaining structure matrices, SO^k and HO^k are stored in a sparse format.

3.5. Simulation algorithm

The parallel simulation algorithm is locally implemented at each logical process. As we already explained, each LP contains an *Integrator*, a *Quantizer*, and a *Model* module. The *Quantizer* and *Model* modules are identical to their sequential counterparts, while the *Integrator* now includes the communication and synchronization mechanisms.

The communication involves informing the new values of states and discrete variables to other LPs after state changes or event handler executions that, according to the inter-process structure, affect state derivatives or zero-crossing functions calculated at other LPs. On the other side, the LPs must check if those new values were changed.

As we explained before, the synchronization mechanism limits the advance of the simulation at each process in order to keep a bounded difference between the logical simulation times of the different LPs.

Later on, we shall explain in more detail the communication and synchronization procedures.

Taking into account these modifications, the parallel kth *Integrator* algorithm works as follows:

Algorithm 5: QSS Parallel Integrator Module

```
1  while t^k < t_f //while simulation time of the k--
       th LP is less than the final time tf
2    t_x^k = min(tx_j|PX_j = k) // time of the next change
         in a quantized variable computed by
         the k-th LP
3    t_z^k = min(tz_j|PZ_j = k) // time of the next zero--
         crossing time computed by the k--th LP
4    t_m^k = ChangesList.First().time() // timestamp
         of the first informed change
5    t^k = min(t_x^k, t_z^k, t_m^k) // attempt to advance LP
         simulation time
6    gvt = min(t^l) // recompute global virtual
         time
7    Synchronize() // call synchronization
         procedure
8    if t^k = t_x^k then //state change
9      i = argmin(tx_j) // the i-th quantized state
           changes first
10     Parallel_Quantized_State_Step(i) //
           execute a quantized state change on
           variable i
11   elseif t^k = t_z^k then //zero--crossing
12     i = argmin(tz_j) // the i-th event handler
           is executed first
13     Parallel_Event_Handler_Step(i) //execute
           the procedure for the i-th event
           handler
14   else
15     Process_External_Change() //process the
           first enqueued change
16   end if
17 end while
```

This algorithm is similar to its sequential counterpart (Algorithm 2). It adds the calculation of the global virtual time gvt as the minimum between the local simulation times of all LPs and a synchronization routine to avoid that the local time t^k advances much beyond gvt. Additionally, it takes into account the incoming changes in the input variables informed by different LPs.

When the next step in the kth LP corresponds to a change in a quantized variable q_i at time t, the *Integrator* proceeds as follows:

Algorithm 6: QSS Parallel Integrator Module — Quantized State Change

```
1  Parallel_Quantized_State_Step(i)
2  {
3    integrateState(x_i, ẋ_i, t^k) // integrate i-th
         state up to time t^k.
4    Quantizer.update(x_i, q_i) // update i-th
         quantized state q_i
5    tx_i = Quantizer.nextTime(x_i, q_i) //compute next
         i-th quantized state change time
6    for each l such that SO^k_{i,l} = 1
7        ChangesList.InsertState(l,i,q_i,t_k) //
             inform new quantized state and time
             stamp to l--th LP
8    end for
9    for each j such that SD_{i,j} = 1 and PX_j = k
10     integrateState(x_j, ẋ_j, t) // integrate j-th
           state up to time t.
11     ẋ_j = Model.f_j(q^k(t), d^k(t), t) // recompute j-th
           state derivative
12     tx_j = Quantizer.nextTime(x_j(t), q_j(t)) //
           recompute next j-th quantized state
           change time
13   end for
14   for each j such that SZ_{i,j} = 1 and PZ_j = k
15     zc_j = Model.zc_j(q^k, d^k(t), t) // recompute j-th zero
           --crossing function
16     tz_j = nextEventTime(zc_j) //recompute next j-th
            zero--crossing time
17   end for
18 }
```

This algorithm is very similar to the sequential one (Algorithm 3). The difference is that it only updates the state derivatives that are calculated by the current LP, and, additionally, it informs the values and the corresponding time-stamp to the other LPs that compute the remaining state derivatives affected by the quantized state change.

Notice that the entire quantized state vector $\mathbf{q}(t)$ used to calculate state derivatives and zero-crossing functions can contain components $q_j(t)$ computed at different LPs. In that case, the kth LP has a *local* copy (possibly outdated) of its actual value. Thus, the vector is denoted as $\mathbf{q}^k(t)$, comprising the components calculated at the LP and the local copies of the remaining entries. A similar remark can be done regarding the discrete state $\mathbf{d}(t)$.

When the next step corresponds to the execution of an event handler H_i at time t, the *Integrator* proceeds like Algorithm 4 with identical modifications.

As we mentioned above, when a Logical Process informs changes in quantized states or discrete variables, the new values are stored in a time-stamp-sorted list. Additionally, on each simulation step, every LP checks the time-stamp of the first element of the list and, in case the time-stamp ($m_1.t$) is less than the time of the scheduled next local change, the *Integrator* processes the external change as follows (assuming that the incoming new values are provoked by a quantized state change):

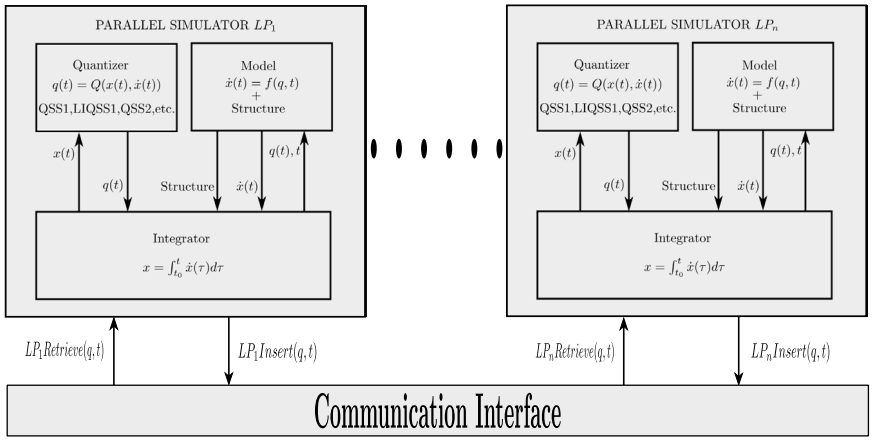

Fig. 2. Parallel stand alone QSS solver — basic interaction scheme.

Algorithm 7: QSS Parallel Integrator Module — External Quantized State Change

```
1  Process_External_Change()
2  {
3    m1=ChangesList.RetrieveState() //get the
         first element of the list.
4    i=m1.index() //index of the quantized variable
          that changed
5    q_i^k = m1.q //update local copy of quantized
         state q_i.
6    for each j such that SD_{i,j} = 1 and PX_j = k
7      integrateState(x_j, ẋ_j, t) // integrate j-th
           state up to time t.
8      ẋ_j = Model.f_j(q(t), d(t), t) // recompute j-th state
           derivative
9      tx_j = Quantizer.nextTime(x_j(t), q_j(t)) //
           recompute next j-th quantized state
           change time
10   end for
11   for each j such that SZ_{i,j} = 1 and PZ_j = k
12     zc_j = Model.zc_j(q, d(t), t) // recompute j-th zero
           --crossing function
13     tz_j = nextEventTime(zc_j) //recompute next j-th
           zero--crossing time
14   end for
15 }
```

When the incoming variables that changed corresponds to a handler execution, the procedure is similar to the previous one.

3.6. Inter-process synchronization

As we mentioned before, the parallel implementation of the QSS solver uses a non-strict synchronization mechanism that is necessary to ensure that calculations are actually performed in parallel.

This synchronization mechanism requires that, before each simulation step, the LPs calculate

$$gvt = \min_{1 \le l \le P} (t^l) \tag{8}$$

as the minimum logical simulation time of all LPs. Then, each LP limits the advance of its local simulation time t^k to the value $gvt + \Delta t$, where Δt is a user defined parameter. Notice that Δt is an upper bound for the difference between the local simulation times of all the LPs and it defines how strict the synchronization is.

The synchronization procedure of each LPs implements a waiting routine while the condition

$$t^k > gvt + \Delta t \tag{9}$$

is verified. When gvt advances (because other LPs update their local simulation time) or when t^k goes back (because the kth LP detected an external change), and the condition of Eq. (9) is no longer accomplished, the simulation at the kth LP continues. That way, the synchronization routine works as follows:

Algorithm 8: LP Synchronization

```
1  Synchronize()
2  {
3    while(t^k − gvt > Δt)
4      rcv = ChangesList.First().time() //
           timestamp of the first informed change
5      if (rcv < t^k)
6        t^k = rcv
7      gvt = min(t^l) // recompute the minimum global
           virtual time
8    end while
9  }
```

Notice that the line 7 of this code explicitly computes the minimum global virtual time depending on the local logical times t^l of all processes. This calculation is the reason that restricts the usage of the whole strategy to shared memory architectures. Any attempt to use this algorithm on a distributed memory architecture would result in a huge traffic through the network communicating the logical times of the different processors at each step.

Fig. 3 illustrates the synchronization mechanism during a parallel simulation with two LPs.

There, the maximum simulation time to which a LP can advance is limited by $gvt + \Delta t$, represented by the gray region above gvt.

At the beginning of the simulation, the second logical process (LP2) schedules its next change t^2 to a value that is larger than $gvt + \Delta t$, so it must wait until gvt advances (i.e., the simulation time t^1 of LP1 advances).

Then, at the instant of CPU time t_a, the simulation time t^1 (and also gvt) is such that $t^2 < gvt + \Delta t$ and LP2 can resume the simulation steps. During the interval (t_a, t_b), the distance between t^1 and t^2 is less than Δt and so both processes simulate in parallel without waiting for synchronization. Then, at time t_b, the first process LP1 schedules its next simulation step for time $t^1 > gvt + \Delta t$, and now it must wait until the synchronization point is reached again.

Notice that if we had taken $\Delta t = 0$, then each LP can only advance when $gvt = t^k$, leading to a sequential simulation and there is no parallelization at all. On the contrary, if we set $\Delta t \ge t_f$ (where t_f is the final simulation time) there is no synchronization

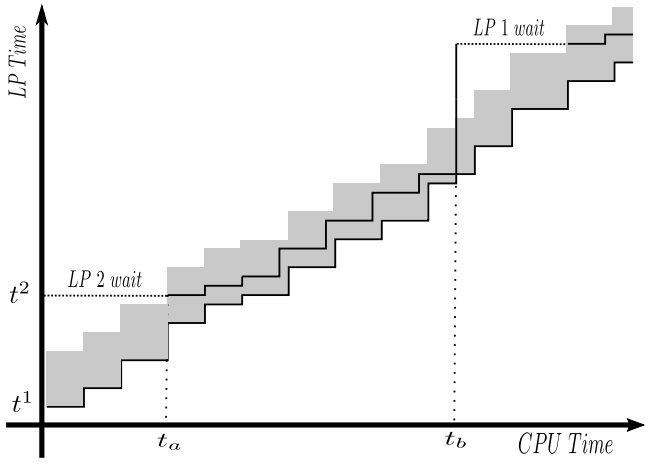

Fig. 3. Parallel QSS solver synchronization scheme example.

at all and each LP simulates as fast as it can, like in a *No-Time* strategy, but the numerical error introduced would possibly result unacceptable.

Thus, a good value for Δt should be such that the error is acceptable and enough parallelization is achieved. We shall deeply analyze this problem later, proposing also an adaptive strategy for finding an adequate value for Δt.

3.7. Inter-process communication

As we mention above, the different LPs informs changes in state and discrete variables during the simulation. For that purpose, each logical process has a list of the changes occurred in the variables computed at the other LPs that are needed for the local computations. This list is sorted by their corresponding time-stamps and it provides the following services:

- `InsertState(l,i,qi,tk)`: Inserts the new value of the ith quantized state q_i that changed at time t_k computed by the kth LP into the lth LP list of changes.
- `InsertEvent(l,i,dj,xj,tk)`, Inserts the new values of the set of discrete states $\{d_j\}$ and the set of continuous states $\{x_j\}$ that changed after the execution of the ith event handler at time t_k computed by the kth LP into the lth LP list of changes.
- `RetrieveState()`, Gets the first element of the list of changes caused by an external quantized state update.
- `RetrieveEvent()`, Gets the first element of the list of changes caused by an external handler execution.

If a change with a time-stamp $m.t$ less than the time of the last step performed by the LP t^{k-} is detected, then the time-stamp is modified to t^{k-}. That way, we avoid that the simulation goes back in time.

Notice that the modification of this time-stamp is the only effect that synchronization errors have on the simulation results. Anyway, the difference between $m.t$ and t^{k-} is always bounded by Δt.

The communication mechanism between threads is hybrid:

- State changes are asynchronously informed using the previously mentioned lists of changes.
- Discrete changes are also informed using lists of changes, but in a synchronous way where the LP that communicates the new values must wait until all the receiver LPs process them.

The later case ensures that the new values of all the discrete variables are processed by all the affected LPs before the LP that is responsible for the change continues with the simulation. Taking into account that discrete changes can provoke large changes in discrete and state variables, this policy prevents the introduction of large numerical errors due to lack of synchronization after discontinuities.

It is worth mentioning that in most models containing discontinuities the number of continuous changes is significantly larger than the number of discrete changes. Therefore, in practice, the communication is mostly asynchronous.

3.8. Some implementation details

The execution model for each logical processor is implemented using native Linux POSIX threads. Threads are pinned to specific CPUs using `sched_setaffinity`. The assignment of threads to CPUs is given by the partition matrices *PX* and *PZ*.

Initially, the main thread reads the partition structure and computes the interprocess structure matrices. Then, it creates P threads where each of them locally implements an instance of a *Parallel Simulator* module containing a *Parallel Integrator*, a *Quantizer*, and a *Model* modules.

At this point, there are two alternative implementations. The first one, uses a *Model* module identical to that of the sequential implementation. Since the sequential model computes the state derivatives, handlers and zero crossing functions depending on the full state, this implementation requires working with arrays containing the full state size.

In this first implementation, the *Parallel Integrator* module has a data structure containing the following local data:

- The local logical simulation time t^k and a local copy of the global virtual time gvt.
- The corresponding inter-process structure matrices SO^k and HO^k.
- Two arrays containing the indexes of the state derivatives and zero-crossing functions evaluated at the current LP.
- A sorted list containing changes corresponding to quantized state variables.
- A sorted list containing changes corresponding to event handler executions.

Additionally, a compilation flag allows to keep local copies of

- The full state and quantized state arrays **x**, and **q**.
- The full array of zero-crossing function values.
- The full discrete state array **d**.

Otherwise, those arrays are shared by all the LPs together with the remaining global data, consisting of the following:

- An array containing the local simulation times of all LPs t^l.
- The four incidence matrices *SD*, *HD*, *SZ*, and *HZ*.

The optional usage of local copies for the full state and quantized state allows to target Non-Uniform Memory Access (NUMA) platforms. Here, the memory latency for a given address depends on each processor. Thus, by allocating local copies we enforce that each processor accesses its own memory bank. Independently on this, having local copies of the full discrete array can help to avoid concurrency problems due to the fact that each discrete variable can be computed by more than one LP in certain models.

The price paid for using local full array copies is a large increase in the memory consumption. However, it simplifies the implementation, and, in cases where the memory usage in not critical, acceptable results can be obtained.

A second implementation uses a different model instance for each LP, so that the functions defining each sub-models depend only on the local variables of each LP. Also, the structure information provided by each sub-model is local to that sub-model. Thus, in this implementation, each LP contains only local copies

of the corresponding sub-arrays, whose size is proportional to the size of the sub-model. That way, as we increase the number of partitions, the total memory consumption is almost unaffected.

The drawback of this implementation is that it requires the generation of a different model instance for each LP. Starting from the original model and structure matrices, the construction of the partitioned sub-models requires remapping the state variable indexes and to generate the corresponding code for the resulting functions and structure. In theory, this problem has a straightforward solution, but its automatic implementation requires to modify the *Model Generator* module of the QSS Solver.

We are currently working on this last issue. So far, the current version of the parallel QSS Solver supports the usage of partitioned models, but the sub-model instances must be manually generated.

The performance of the alternative implementations shall be compared later on an example in Section 6.1.

For both implementations, in spite of the fact that all the data is allocated in shared memory, the only data that is actually updated by the different processes concurrently are the arrays containing the local simulation times of the LPs (for synchronization purposes) and the changes lists used for communication.

The array with the local simulation times is accessed for computing the global virtual time gvt. Anyway, in the implementation, gvt is only updated at the kth LP when the local time t^k becomes greater than $gvt + \Delta t$. That way, in most steps, there is no actual access to that shared array.

4. Synchronization and numerical errors

The non strict synchronization mechanism adopted can provoke that when the kth LP informs a change in a quantized state q_i at time t^k, this change is detected late by the lth LP, i.e., when the lth LP has already performed one or more steps at time $t^l > t^k$. In that case, as there is no roll-back mechanism, the lth LP will just modify the time-stamp of the incoming values pretending that the change in q_i occurred at time t^l.

This implies that, during the interval (t^k, t^l), the lth LP integrates using an incorrect (outdated) local copy of the quantized state q_i. Anyway, taking into account that the difference between t^l and t^k is bounded by Δt, and knowing that q_i is an approximation of a continuously varying signal $x_i(t)$, we can expect that the difference between the actual value of q_i (computed at the kth LP) and the local copy at the lth LP is not significant and that it will introduce a bounded numerical error in the subsequent calculations.

The following analysis formally shows this fact.

Consider the QSS approximation of an ODE, given by

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), t)$$

and suppose that this system is partitioned in two parts as follows:

$$\begin{aligned} \dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q}_a(t), \mathbf{q}_b(t), \mathbf{v}(t)) \\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q}_a(t), \mathbf{q}_b(t), \mathbf{v}(t)) \end{aligned} \tag{10}$$

where

$$\mathbf{x} = [\mathbf{x}_a\ \mathbf{x}_b]^T; \qquad \mathbf{q} = [\mathbf{q}_a\ \mathbf{q}_b]^T; \qquad \mathbf{f} = [\mathbf{f}_a\ \mathbf{f}_b]^T.$$

Let us consider that we implement the simulation of this system in parallel, such that $\mathbf{x}_a$ is computed in one LP and $\mathbf{x}_b$ is computed in a second LP according to the mechanism explained in the previous section.

Then, the non strict synchronization implies that the first LP can have a local copy $\mathbf{q}_b^c(t)$, which is an outdated version of the components of $\mathbf{q}_b(t)$. Similarly, the second LP can have a local copy $\mathbf{q}_a^c(t)$, containing outdated versions of the components of $\mathbf{q}_a$.

Thus, the parallel algorithm simulates the following approximation of Eq. (10):

$$\begin{aligned} \dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q}_a(t), \mathbf{q}_b^c(t), \mathbf{v}(t)) \\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q}_a^c(t), \mathbf{q}_b(t), \mathbf{v}(t)). \end{aligned} \tag{11}$$

Let us define $\Delta_a(t) \triangleq \mathbf{q}_a^c(t) - \mathbf{q}_a(t)$, and $\Delta_b(t) \triangleq \mathbf{q}_b^c(t) - \mathbf{q}_b(t)$. Then, Eq. (11) can be rewritten as

$$\begin{aligned} \dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q}_a(t), \mathbf{q}_b(t) + \Delta_b(t), \mathbf{v}(t)) \\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q}_a(t) + \Delta_a(t), \mathbf{q}_b(t), \mathbf{v}(t)) \end{aligned} \tag{12}$$

that constitutes a *perturbed* version of the original system of Eq. (10), with perturbation terms $\Delta_a(t)$ and $\Delta_b(t)$.

Notice that a component $q_i(t)$ computed at the first LP and its local copy $q_i^c(t)$ at the second LP can have a maximum delay in time given by the parameter Δt, i.e., $q_i^c(t) = q_i(t-\tau)$ with $0 \leq \tau < \Delta t$. Then, it results that

$$|q_i^c(t) - q_i(t)| = |q_i(t-\tau) - q_i(t)| \leq |x_i(t-\tau) - x_i(t)| + 2\Delta Q_i$$

where we used the fact that the difference between x_i and q_i is always bounded by the quantum ΔQ_i. Then, assuming that the state derivative $\dot{x}_i = f_i(\mathbf{q}, t)$ is bounded by a constant M_i while the quantized states $\mathbf{q}(t)$ remains in certain bounded region, it results that

$$|x_i(t-\tau) - x_i(t)| < M_i \cdot \tau \leq M_i \Delta t$$

and then

$$|q_i^c(t) - q_i(t)| \leq M_i \cdot \Delta t + 2\Delta Q_i. \tag{13}$$

Applying the same analysis to all the components of the array $\mathbf{q}_a$ and $\mathbf{q}_b$, it results that all the components of the perturbation terms $\Delta_a(t)$ and $\Delta_b(t)$ are bounded in their absolute value for a constant depending on the parameter Δt and the quantum ΔQ.

When the original system of Eq. (1) is a linear time invariant (LTI) and asymptotically stable, it can be easily proved that the presence of the bounded perturbations $\Delta_a(t)$ and $\Delta_b(t)$ only adds a bounded numerical error proportional to the perturbation bound [24,8]. In nonlinear cases, assuming that the perturbation bound is small enough, a similar property can be derived [28].

If we had more than two processors, the analysis can be easily extended arriving to the same results.

In conclusion, the parallelization introduces an additional error to that introduced by the QSS methods. This additional error can be bounded by a quantity that increases proportional to the parameter Δt.

Although this conclusion is of qualitative nature (i.e., it does not provide a quantitative bound for the additional numerical error), in the next section we shall derive an algorithm for adaptive computation of Δt in order to ensure a quantitative error bound.

5. Adaptive Δt adjustment

The election of the parameter Δt imposes a trade-off between numerical errors and the actual parallelization achieved. Obtaining a suitable value for Δt usually involves running a set of experiments in order to observe the error introduced by the parameter and the speedup obtained. Also, a value for Δt that is suitable at the beginning of the simulation may be not optimal after some time due to changes in the dynamics of the model.

In order to overcome these problems, we developed an adaptive algorithm that allows the dynamic calculation of Δt in order to keep a bounded numerical error.

As we analyzed in the previous section, the parallel implementation with non strict synchronization provokes a difference between the quantized states computed in one LP and their local

copies used in other LPs. This difference can be seen as a bounded perturbation that introduces a bounded numerical error.

The difference between a quantized state $q_i(t)$ and a local copy $q_i^c(t)$ is bounded according to Eq. (13). Thus, if we want to bound this perturbation to be proportional to the prescribed tolerance, it should be proportional to the quantum ΔQ_i:

$$|q_i^c(t) - q_i(t)| \leq M_i \cdot \Delta t + 2\Delta Q_i \leq \alpha \cdot \Delta Q_i.$$

Here, M_i is an upper bound for the state derivative $\dot{x}_i(t)$ and α is a user defined parameter. From this last inequality, we obtain

$$\Delta t \leq \frac{(\alpha - 2) \cdot \Delta Q_i}{M_i}. \tag{14}$$

That limit on Δt must be accomplished for all the quantized states that have local copies in other LPs, i.e, for all the *output* quantized states.

The set of output quantized states of the kth LP can be formally defined as $O_k \triangleq \{i | \exists l \text{ such that } SO_{i,l}^k = 1\}$. Then, the set of all output quantized states results $O \triangleq \bigcup_k O_k$.

As we want that Eq. (14) is satisfied for all the output quantized states, the adaptive algorithm takes Δt as

$$\Delta t = \min_{i \in O} \frac{(\alpha - 2) \cdot \Delta Q_i}{M_i}. \tag{15}$$

For that goal, each LP computes its minimum Δt^k as

$$\Delta t^k = \min_{i \in O_k} \frac{(\alpha - 2) \cdot \Delta Q_i}{M_i} \tag{16}$$

and then Δt is computed as the minimum of all Δt^k.

The bounds on the state derivatives M_i are numerically estimated as[1]

$$M_i \approx \frac{|x_i(t_i) - x_i(t_i^{prev})|}{t_i - t_i^{prev}} \tag{17}$$

where t_i and t_i^{prev} are the two last times of change in the quantized state q_i.

Notice that the estimate M_i is updated each time the output quantized state q_i changes. That change in M_i may modify the LP minimum Δt^k computed in Eq. (16) that can in turn change the global Δt. Thus, Δt is asynchronously updated. This fact in turn implies the need of an additional synchronization mechanism that must be introduced in the *Parallel Integrator* module.

To achieve this goal, the *Parallel Integrator* Algorithm 5 must be modified so that when a simulation step changes an output quantized variable, the local minimum Δt^k can be updated. If so, the algorithm checks if this update changes the global value Δt. If the global parameter is affected, a global synchronization routine is called where all LPs obtain the new value for Δt.

For that purpose, the data structure of each LP is extended with the addition of a local copy of the global Δt parameter. Also, a global array shared by all LPs contains the local minimum Δt^k.

Notice that the usage of this strategy only requires to choose the *error bound coefficient* α, and then the parameter Δt is automatically adapted. In principle, the choice $\alpha \approx 10$ seems reasonable, as the error introduced by the lack of synchronization would be at most one order of magnitude larger than the error introduced by the QSS approximation itself.

Notice that, while the analysis of Section 4 provides a qualitative error bound, the strategy for adaptation of Δt provides a quantitative bound for the numerical error, where we ensure that it cannot become larger than α times the error introduced by the sequential QSS algorithm. Thus, provided that the quantum ΔQ was adequately chosen by the user for the sequential case, the parallel implementation will work within a prescribed tolerance.

6. Results

In this section, we present the simulation results on four large-scale models. In these models, we analyze and compare the performance of the parallel QSS Solver using the proposed synchronization strategies with different parameter settings.

The examples are ordered by their complexity. The first one, corresponding to a large population of air conditioner units, is a hybrid system that does not require communication between LPs so it is useful to measure the overhead introduced by the Parallel QSS Solver mechanism. The second model is a two dimensional Advection–Reaction partial differential equation (PDE) semi-discretized with the Method of Lines [8]. This is a purely continuous system that allows also to validate the results regarding the numerical error introduced by the parallelization strategies. The third model is a modification of the first one with the addition of a centralized power control, that introduces inter-process communication of discontinuous changes. The last one, corresponding to a network of spiking neurons, has a complex connection structure with events that provoke instantaneous changes on continuous states.

Simulation framework

The results presented below were obtained using a 64-core server with 32 GB of RAM running a 64 bit Linux OS (Ubuntu 14.04). The server is composed of 4 AMD Opteron 6272 processors, with 16 cores (in 8 physical modules) each[2] and a Non-uniform memory access (NUMA) architecture. We used the Parallel QSS Solver implementation version 3.1 (git commit number [$c10a14$]).

In order to evaluate the performance of the simulations we took into account the following metrics:

- Initialization Time: It corresponds to the time spent at the initialization routines of the Parallel QSS Solver.
- Simulation Time: It is the CPU time taken by the simulation at the slowest logical process. It does not take into account the Initialization Time.
- Memory Usage: We report the total amount of memory allocated by the simulation.
- Simulation Speedup: $Speedup(P) = \frac{T_1}{T_P}$, where T_1 is the simulation time spent using a single core and T_P is the simulation time spent using P cores.
- Normalized Mean Error[3]:

$$error = \frac{mean(|\mathbf{y}^P - \mathbf{y}^{ref}|)}{mean(\mathbf{y}^{ref})} \tag{18}$$

where the reference trajectory $\mathbf{y}^{ref}$ is the one obtained using a sequential simulation whereas $\mathbf{y}^P$ is the trajectory obtained

[1] M_i can be directly obtained as $M_i = |\dot{x}_i|$, as $\dot{x}_i$ is stored by the QSS algorithms. However, when a solution oscillates around an equilibrium point (as it occurs often in QSS solutions) $\dot{x}_i$ can be larger than the actual variation of the state, and a numerical approximation can provide a better result.

[2] The AMD Opteron 6272 processors are based on the AMD Bulldozer micro-architecture. They have two integer units but only one floating point (FP) scheduler per module, thus, depending on the application, each processor can be considered to have 8 or 16 cores. As the QSS solver is always combining FP with integer operations (as the structure plays a fundamental role in the computations), it can be expected in principle that the 16 threads can efficiently run in parallel.

[3] In this work, we compute the mean error rather than the maximum global error because in the examples the states are either discontinuous or have very steep trajectories. That way, a very small delay between two trajectories provokes that the maximum error is equal to the trajectory amplitude (independently on the delay value). Thus, the mean error captures better the actual difference between the simulated trajectories.

with P cores. Both trajectories are evaluated at 5000 equidistant time points.

Notice that this error does not include the numerical error introduced by the QSS approximation. It only evaluates the additional numerical error introduced by the lack of synchronization between processes.

For the different models, we performed the following experiments:

1. In order to analyze the effects of using different values of the synchronization parameter Δt, we run several simulations varying that parameter. These simulations were performed using 62 cores.[4]
2. After selecting a suitable value for Δt (i.e., a value that achieves a good trade-off between speedup and error) we run simulations varying the number of cores in order to characterize the speedup ratio w.r.t. the number of cores.
3. For the adaptive synchronization strategy, we run several simulations varying the error bound parameter α on a 62 cores configuration.
4. Then, using the error bound parameter $\alpha = 10$ (which is a reasonable choice, as analyzed in the previous section), we run simulations varying the number of cores in order to characterize the achieved speedup.

In all cases, the models were manually partitioned.

6.1. Air conditioners population

This model, taken from [39], studies the dynamics of a large population of air conditioners (ACs) when they follow the same reference temperature. The ith air conditioner is used to control the temperature of the ith room, modeled by:

$$\dot{\theta}_i(t) = -\frac{1}{C_i \cdot R_i}[\theta_i(t) - \theta_a + R_i \cdot P_i \cdot m_i + w_i(t)] \tag{19}$$

where R_i represents the thermal resistance, C_i is the thermal capacity, P_i is the power of the AC unit in *on* state, θ_a is the outside ambient temperature (common to all rooms), and $w_i(t)$ is a noise signal representing thermal disturbances.

The variable $m_i(t)$ is the state of the ith AC unit that takes the value 1 in *on* state and 0 otherwise. This state follows a hysteretic on–off control law:

$$m_i(t^+) = \begin{cases} 0 & \text{if } \theta_i(t) \leq \theta_r^i(t) - 0.5 \text{ and } m_i(t) = 1 \\ 1 & \text{if } \theta_i(t) \leq \theta_r^i(t) + 0.5 \text{ and } m_i(t) = 0 \\ m_i(t) & \text{otherwise} \end{cases} \tag{20}$$

where $\theta_r^i(t)$ is the reference temperature, that obeys the following profile:

$$\theta_r^i(t) = \begin{cases} 20 & \text{if } 0 \leq t < 1000 \\ 20.5 & \text{if } 1000 \leq t < 2000 \\ 20 & \text{if } 2000 \leq t \leq 3000. \end{cases} \tag{21}$$

Finally, the thermal disturbances $w(t)$ is updated once every minute, taking pseudo-random values uniformly distributed in the interval $(-1, 1)$. The temperature of the first AC unit is depicted in Fig. 4.

For this experiment, we simulated a population of 248,000 AC units and set their parameters according to [39].

Notice that each air conditioner is modeled by one differential equation (Eq. (19)) and three zero crossing functions: one associated to the hysteretic control law of Eq. (20), another corresponding to the evolution of the reference temperature of Eq. (21), and the last one associated to the update of the noise signal $w_i(t)$. Thus, the model contains a set of 248,000 differential equations and 992,000 zero-crossing functions.

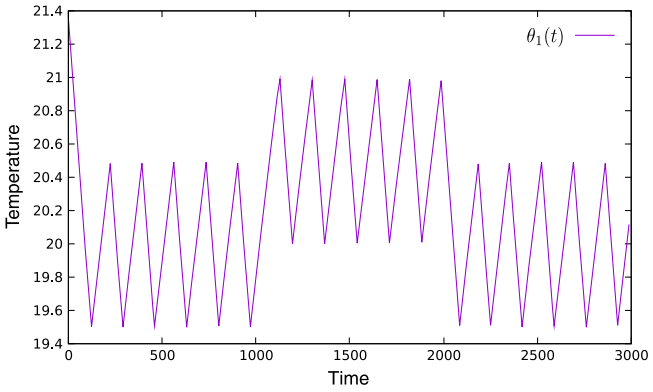

Fig. 4. Air conditioners population model — output trajectory.

Observe also that the dynamics of each air conditioner unit is independent on the evolution of the remaining air conditioners. Therefore, the model can be partitioned in a way such that no communication between the different sub-models is required and hence, the lack of synchronization in the parallel simulation does not introduce errors.

This model was simulated using the QSS2 method with a quantization of $\Delta Q_{rel} = \Delta Q_{min} = 1e-3$ and the final simulation time t_f was set to 3000 min.

In this case, the Parallel QSS solver automatically detects that there is no communication between the different LPs and it automatically sets the parameter Δt equal to the final simulation time 3000. Thus, in this introductory case, we did not run experiments for different values of Δt or α as these parameters are meaningless here.

Also, we do not report errors as they were null in all cases (in absence of communication, sequential and parallel results are identical).

The goal of this example is to measure the overhead introduced by the parallel simulation when the synchronization mechanism is not needed. This overhead includes the access to the structural matrices (that are shared by the different LPs), the additional calls to the synchronization routines and the access to the inter-process communication structures. Additionally, in this experiment, we include the results obtained using the model partitioning strategy described in Section 3.8 to improve the memory consumption. In this simple case, we were able to generate the different *Model* instances manually.

Then, we run simulations for a different number of cores ranging from 1 to 62. The results are reported in Table 1.

As we can see in Table 1, the achieved speedups for this model are linear. Running on 62 cores speeds up the simulation by a factor of 45 using full state copies and 49 using the model partition strategy. The speed difference between these strategies can be explained by the additional memory accesses needed by the non partitioned implementation. Initialization times are increased with the number of cores (the initialization involves the computation and allocation of interprocess communication matrices, that grow in complexity with the number of processes). Anyway, the initialization times are neglectable in all cases.

The memory consumption, in turn, grows almost linearly with the number of cores in the non partitioned implementation, increasing the memory consumption more than 28 times for 62 cores. This is due to the fact that each LP has local copies of the states, quantize states, etc., and, additionally, they allocate communication data structures and the interprocess communication

[4] We used a maximum of 62 cores leaving 2 cores free to attend the OS tasks.

Table 1
Air conditioners population model — simulation results speedup and memory consumption comparison between the solver implementation and the model partition prototype (MPP).

Cores	Init time (ms)	Simulation time (s)	Memory (MBytes)	Speedup	MPP simulation time (s)	MPP memory (MBytes)	MPP seedup
1	580	1132	360	1.00	1132	360	1.00
2	894	668	655	1.69	644	388	1.75
4	948	357	1,081	3.17	341	501	3.31
8	969	192	1,835	5.89	185	544	6.11
16	1085	109	3,342	10.38	96	628	11.79
32	1165	52	6,356	21.76	50	804	22.64
48	1281	32	9,371	35.37	31	976	36.51
62	1406	25	10,223	45.28	23	1057	49.21

matrices. However, when using the model partitioning strategy, we can observe that the memory consumption for 62 cores is less than 3 times the memory needed for running the simulation with 1 core. This is due to the fact that each sub-model contains only the sub-array of the state, quantized state and zero-crossing functions that each LP needs, together with the corresponding structural information which is also partitioned. Also, as a consequence, there is no need to store the additional arrays containing the indexes of the state derivatives and zero-crossing functions evaluated at each LP.

As an additional test, we measured the simulation time spent by the sequential implementation of the QSS Solver. This experiment took 1050 s, which is about 8% faster than the parallel implementation running on a single core. This overhead is due to the extra control mechanisms of the parallel solver (for communication, synchronization, and access to inter-process structures). These mechanisms are not disabled when running on a single core although they are not actually necessary. Similar tests were run on the different examples arriving always to very similar overhead figures.

A final remark is that, in this case, the simulation using one core takes less than 20 min so the large-scale nature of the model can be questioned. Anyway, this was just an illustrative case, where we can easily simulate the system until a final time ten times larger arriving to identical conclusions.

6.2. Advection–reaction equation

The following model, taken from [4], represents a 2D Advection–Reaction equation. This equation can describe, for instance, a river transporting a substance experiencing a chemical reaction.

After applying the method of lines on this PDE, the following set of ODEs is obtained:

$$\dot{u}_{i,j} = -a_x \frac{u_{i,j} - u_{i,j-1}}{\Delta x} - a_y \frac{u_{i,j} - u_{i-1,j}}{\Delta y} + r u_{i,j}(u_{i,j} - \alpha)(u_{i,j} - 1) \quad (22)$$

for $i = 2, \ldots, N, j = 2 \cdots M$, where $u_{i,j}(t)$ is the concentration of the transported substance at the i, j grid point of the spatial domain. The parameters a_x and a_y represent the speed of the transporting flow in the x and y coordinates, respectively, and r is the rate of the chemical reaction. Finally, Δx and Δy are the widths of each grid section.

At the borders, the dynamics is defined by

$$\dot{u}_{i,1} = -a_x \frac{u_{i,1}}{\Delta x} - a_y \frac{u_{i,1} - u_{i-1,1}}{\Delta y} + r u_{i,1}(u_{i,1} - \alpha)(u_{i,1} - 1) \quad (23)$$

for $i = 2, \ldots, N$

$$\dot{u}_{1,j} = -a_x \frac{u_{1,j} - u_{1,j-1}}{\Delta x} - a_y \frac{u_{1,j}}{\Delta y} + r u_{1,j}(u_{1,j} - \alpha)(u_{1,j} - 1) \quad (24)$$

for $j = 2, \ldots, M$, and,

$$\dot{u}_{1,1} = -a_x \frac{u_{1,1}}{\Delta x} - a_y \frac{u_{1,1}}{\Delta y} + r u_{1,1}(u_{1,1} - \alpha)(u_{1,1} - 1). \quad (25)$$

Table 2
Advection–reaction model – fixed Δt – using 62 cores.

Δt	Simulation time (s)	Speedup	Error
1e−07	1104	2.40	3.61e−06
1e−06	288	9.20	3.45e−06
1e−05	145	18.28	3.66e−06
1e−04	119	22.27	6.22e−06
1e−03	121	21.90	1.38e−05
1e−02	115	23.05	2.12e−05
1e−01	112	23.66	2.39e−04

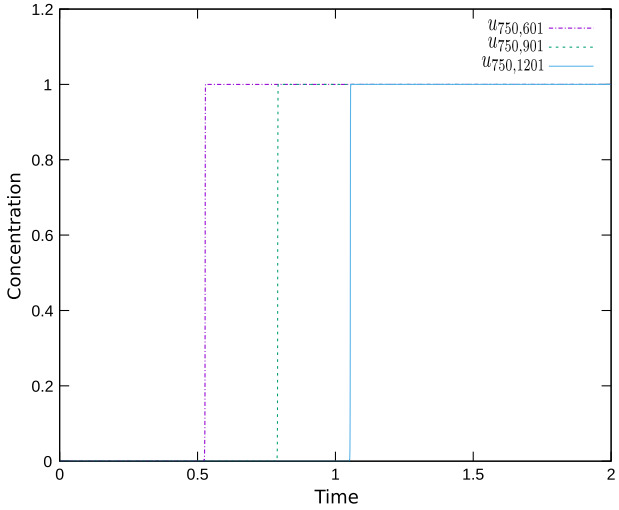

Fig. 5. Advection–reaction model — output trajectories.

The initial conditions are set to

$$u_{i,1} = 1 \quad (26)$$

for $i = 1, \ldots, N$ and $u_{i,j} = 0$ otherwise.

The concentration of three output variables in the middle of the grid, namely $u_{750,301}$, $u_{750,901}$ and $u_{750,1201}$ are illustrated in Fig. 5.

Taking into account that the model is *stiff* (the chemical reactions are much faster than the transport dynamics), the linearly implicit LIQSS2 algorithm was used, setting the tolerance to $\Delta Q_{rel} = \Delta Q_{abs} = 10^{-3}$. The model parameters were $r = 10{,}000$, $a_x = 1$, $a_y = 0.1$, $\Delta x = 1/N$, $\Delta y = 1/M$ with $N = M = 1500$. That way, we have a spatial domain of size 1×1 with $1500 \times 1500 = 2{,}250{,}000$ grid points, where each grid point represents a state equation.

Table 2 reports the simulation results for different values of the Δt parameter on 62 cores. Here, the error reported is the mean error on 20 state variables $u_{i,j}(t)$ taken over a line at the middle of the spatial domain. Each error was computed using Eq. (18).

As expected, for increasing values of Δt, the speedup and the error increase. From these results, evidently, the best performance is obtained with $\Delta t = 1e-4$. Thus, fixing Δt to this value, we computed the speedup and errors for different number of cores (Table 3).

Table 3
Advection–reaction model — fixed $\Delta t = 1e-04$.

Cores	Init time (ms)	Simulation time (s)	Memory (MBytes)	Speedup	Error
1	174	2650	1,058	1.00	0
2	699	2248	1,626	1.18	3.63e−06
4	666	1216	2,763	2.18	3.66e−06
8	748	715	5,035	3.71	3.33e−06
16	850	375	9,580	7.07	3.14e−06
32	1100	203	14,573	13.05	5.44e−06
48	1300	153	23,647	17.32	4.83e−06
62	1460	119	31,614	22.27	6.22e−06

Table 4
Advection–reaction model – adaptive Δt – using 62 cores.

α	Simulation time (s)	Speedup	Error
2	169	15.68	2.96e−06
10	135	19.63	4.50e−06
20	124	21.37	4.57e−06
50	122	21.72	6.71e−06
100	127	20.87	9.33e−06

We can notice that the error has always the same order of magnitude. As before, the memory consumption and the initialization times grow with the number of cores (the initialization time is again negligible). Finally, the speedup reaches a reasonable value of 22 for 62 cores, growing almost linearly with the number of cores.

We also simulated the model using the adaptive Δt strategy, varying the error bound parameter α, obtaining the results reported in Table 4.

As predicted in Section 5, the best results are obtained around $\alpha = 10$, achieving a similar speedup to that obtained using $\Delta t = 1e-4$, but with less error. Notice that using this strategy the speedup and errors are robust with the choice of the parameter α.

Using the error bound parameter $\alpha = 10$, we then varied the number of cores, obtaining the results reported in Table 5.

Notice that the error does not change significantly and the speed up follows a very similar evolution to that obtained using a fixed parameter $\Delta t = 1e-4$.

It is worth mentioning that obtaining the optimal value $\Delta t = 1e-4$ required performing several experiments, while the value $\alpha = 10$ is just the default choice for the adaptive algorithm.

In order to compare the performance of the parallel QSS Solver with that of a classic parallel ODE solver, we simulated the same model using the parallel extension of the CVODE package called PVODE [7] on the same server, under the same tolerance settings. The PVODE solver was configure to use the OpenMP library and the Adams numerical algorithm (BDF did not work due to the very large size). The results are reported in Table 6.

From the results, we also observe an almost linear speed-up until 48 cores, where it saturates. The QSS solver is about 4.73 times faster than CVODE running on a single core. Then, running on 62 cores, the QSS solver becomes about 6.3 times faster than PVODE.

These figures show that the QSS solver is faster in a sequential simulation and the difference becomes bigger using 62 cores. It is worth mentioning that the QSS solver is not optimized to deal with 2D arrays and the simulation code produced is very inefficient.

Table 6
Advection–Reaction model – CVODE solver – simulation results.

Cores	Simulation time (s)	Speedup
1	12,552	1
2	6,814	1.84
4	3,581	3.50
8	1,854	6.77
16	1,293	9.70
32	909	13.80
48	741	16.93
62	751	16.71

6.3. Power control of an air conditioner population

This model, also taken from [39], extends the model of Section 6.1, by adding a central control of the total power consumption. Towards this end, the total AC population is divided into local sections $S = \{s_1, \ldots, s_n\}$ and the power consumed by the kth section is computed as

$$p_i(t_k) = \sum_{j \in s_k} P_j \tag{27}$$

and the total power consumption of the AC population is computed as

$$P(t_k) = \sum_{i \in S} p_i. \tag{28}$$

The global control system regulates the total power $P(t_k)$ so that it follows a desired power profile $P_r(t)$. To achieve this goal, a proportional integral (PI) control law is used to compute the common reference temperature as follows:

$$\theta_r(t_k) = K_P \cdot [P_r(t_k) - P(t_k)] + K_L \cdot \int_{\tau=0}^{t_k} [P_r(\tau) - P(\tau) d\tau]. \tag{29}$$

Fig. 6 depicts the output of the power control of a sequential simulation of the model.

We simulated a population of 248,000 ACs with 248 local sections, so in this case the model contains 248,000 state variables and 496,498 zero-crossing functions. We used the QSS3 method

Table 5
Advection–reaction model – adaptive Δt – $\alpha = 10$.

Cores	Init time (ms)	Simulation time (s)	Memory (MBytes)	Speedup	Error
1	174	2650	1,058	1.00	0
2	672	2251	1,626	1.18	3.63e−06
4	720	1218	2,763	2.18	3.54e−06
8	733	716	5,035	3.70	3.75e−06
16	871	381	9,580	6.96	3.05e−06
32	1015	207	14,573	12.80	3.70e−06
48	1270	178	23,647	14.89	4.99e−06
62	1468	135	31,614	19.63	4.50e−06

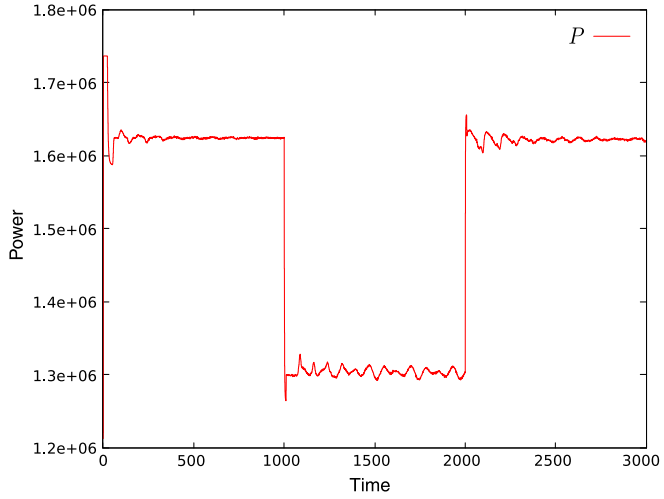

Fig. 6. Power control of an air conditioner population model — output trajectory.

Table 7
Power control of an air conditioner population model — fixed Δt — using 62 cores.

Δt	Simulation time (s)	Speedup	Error
1e−05	2009	1.56	1.42e−04
1e−04	761	4.13	1.82e−04
1e−03	432	7.27	1.81e−04
1e−02	250	12.56	1.85e−04
1e−01	200	15.70	2.95e−04
5e−01	94	33.39	4.99e−04
1	90	34.88	5.50e−04
2	124	25.32	5.84e−04

with a quantization of $\Delta Q_{rel} = 1e-4 = \Delta Q_{min} = 1e-4$ and the model parameters set as in Section 6.1.

The error introduced by the parallel implementation was measured on the total power consumption of the ACs ($P(t_k)$) according to Eq. (18).

Table 7 reports the simulation results obtained for different Δt values using 62 cores.

As expected, the speedup and error grow with the value of Δt. Starting from $\Delta t = 1$, there is no appreciable gain in the speedup, so we used this value for the next experiments. This time, the maximum speedup is in the order of 35 for 62 cores (it was 52 in the first example, that did not need synchronization). In this example, the different LPs communicate discrete changes (changes in the power consumption of each section or changes in the global reference temperature). As explained in Section 3.7 the communication of discrete changes enforces a logical process to wait until the receivers process the corresponding change, what slows down the simulation.

Then, using $\Delta t = 1$, we simulated the model for different number of cores obtaining the results reported in Table 8.

The results are very similar to those of the previous example in terms of initialization times, memory consumption and errors. The speedup, again, grows almost linearly.

Table 8
Power control of an air conditioner population model — fixed $\Delta t = 1$.

Cores	Init time (ms)	Simulation time (s)	Memory (MBytes)	Speedup	Error
1	106	3139	458	1.00	0
2	240	1830	655	1.72	9.18e−04
4	267	1019	1,048	3.08	1.02e−03
8	294	648	1,835	4.84	2.02e−04
16	382	426	3,440	7.37	1.25e−03
32	526	176	6,651	17.84	5.46e−04
48	673	117	9,830	26.83	5.72e−04
62	796	90	12,648	34.88	5.50e−04

Table 9
Power control of an air conditioner population model — adaptive Δt- using 62 cores.

α	Simulation time (s)	Speedup	Error
2	128	24.53	3.88e−04
10	91	34.50	1.47e−03
20	92	34.12	1.46e−03
50	89	35.27	1.39e−03
100	90	34.88	1.59e−03

Then, we simulated the system with the adaptive algorithm for different error bound parameter α, obtaining the results reported in Table 9.

Again, the value $\alpha = 10$ has an almost optimal performance. Using this parameter, we simulated the model for different number of cores, obtaining the results reported in Table 10.

Again, the results are very similar to those obtained with the optimal value $\Delta t = 1$ in Table 7.

6.4. Spiking neural network

The last model, adapted from [45], represents a network of spiking neurons. The ith neuron is modeled by three differential equations:

$$\tau \cdot \dot{v}_i(t) = v_{rest} - v_i(t) + g_i^{ex}(t) \cdot (E^{ex} - v_i(t)) + g_i^{inh}(t) \cdot (E^{inh} - v_i(t))$$
$$\tau^{ex} \cdot \dot{g}_i^{ex}(t) = -g_i^{ex}(t)$$
$$\tau^{inh} \cdot \dot{g}_i^{inh}(t) = -g_i^{inh}(t)$$

where $v_i(t)$ represents the membrane potential, $g_i^{ex}(t)$ is the excitatory conductance, and $g_i^{inh}(t)$ is the inhibitory conductance. The definition and values for the remaining parameters can be found in [45].

Whenever the membrane potential $v_i(t)$ reaches the threshold value −50, the neuron performs a spike, reseting its potential to the value $v_i(t) = v_{rest} = -60$. The spike is transmitted to the post synaptic connected neurons, that change their excitatory or inhibitory conductance depending on the type of neuron that performed the spike. If the ith neuron is of excitatory type, the excitatory conductance of their post-synaptic neurons is changed according to

$$g_j^{ex}(t^+) = g_h^{ex}(t) + \Delta g^{ex}$$

for all $j \in Post_i$ (the set of post synaptic neurons of neuron i). If the ith neuron is of inhibitory type, the inhibitory conductance of their post-synaptic neurons is changed as

$$g_j^{inh}(t^+) = g_h^{inh}(t) + \Delta g^{inh}$$

for all $j \in Post_i$. For these synaptic connections, we used parameters $\Delta g^{ex} = 0.4$ and $\Delta g^{inh} = 1.6$.

After a neuron spike, it enters a refractory period during which it does not change its membrane potential.

We considered a network of 300,000 neurons, with 80% of excitatory type and 20% of inhibitory type randomly distributed.

Table 10
Power control of an air conditioner population model — adaptive $\Delta t - \alpha = 10$.

Cores	Init time (ms)	Simulation time (s)	Memory (MBytes)	Speedup	Error
1	106	3139	458	1.00	0
2	195	1799	655	1.75	1.36e−03
4	209	1032	1,048	3.04	1.43e−03
8	210	651	1,835	4.82	4.77e−04
16	251	426	3,440	7.37	1.33e−03
32	313	172	6,651	18.25	9.01e−04
48	381	117	9,830	26.83	1.09e−03
62	425	91	12,648	34.50	1.47e−03

Table 11
Spiking neurons model — fixed Δt — using 62 cores.

Δt	Simulation time (s)	Speedup	Average spikde
1e−04	2201	2.45	15.80
1e−03	726	7.44	15.92
1e−02	410	13.17	15.92
1e−01	304	17.76	15.93
5e−01	227	23.79	15.87
1e+00	214	25.24	15.84
2e+00	216	25.00	15.62

Each neuron has 200 random post synaptic connections, limited to the 800 closest neurons.

Additionally, there is a set of 60,000 neurons that receive input spikes from an external source during the first 100 mili seconds.

The system has a total of 900,000 state equations, and 660,000 zero crossing functions. Example output trajectories for three neurons are depicted in Fig. 7.

We simulated this system using QSS2 algorithm with a tolerance $\Delta Q_{rel} = \Delta Q_{abs} = 1e-3$ until a final time $t_f = 300$.

This model has a chaotic behavior and a small change in the tolerance settings produces a totally different result if we look at the evolution of an individual neuron. Thus, rather than measuring errors, we measured the activity of the neuron network, counting the number of spikes performed during the simulation.

Due to this chaotic behavior, not only the errors but also the speedups change between different simulation runs under the same parameter settings. Thus, the results reported here are computed as the average over 10 experiments.

As in the previous examples, we started by varying the parameter Δt using 62 cores. The results are reported in Table 11.

As expected, the speedup increased with Δt reaching a maximum of about 25 times. Regarding the number of spikes, it did not experience noticeable changes, so the activity was similar.

Table 12 shows the speedup varying the number of cores using $\Delta t = 0.5$.

The initialization times, memory usage, and speedup show similar evolutions to those of the previous examples, and again, the speedup is almost linear.

The use of the adaptive algorithm varying the error bound parameter α produces the results reported in Table 13

This time, the best results are obtained for $\alpha = 100$. However, due to the chaotic behavior of the model, we cannot estimate the effects of the errors introduced by the lack of synchronization.

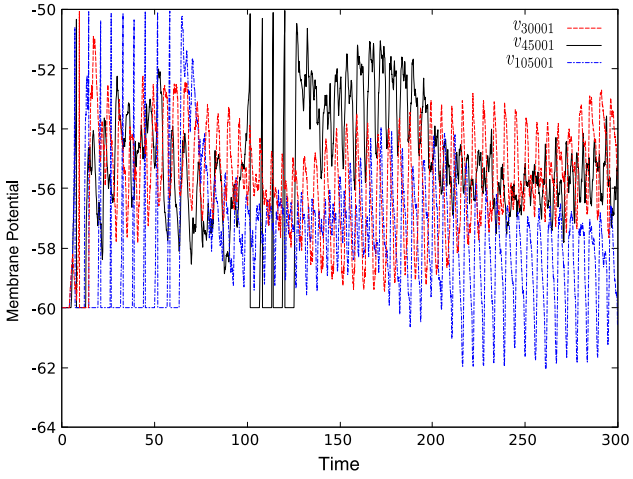

Fig. 7. Spiking neurons model — output trajectories.

Table 13
Spiking neurons model — adaptive Δt- using 62 cores.

α	Simulation time (s)	Speedup	Average spikes
2	388	13.92	15.94
10	344	15.70	15.90
20	315	17.14	16.02
50	252	21.43	15.75
100	236	22.88	15.84

Using $\alpha = 10$ and varying the number of cores, we obtained the results of Table 14.

7. Conclusions

In this article, we presented novel methodologies for the parallel simulation of continuous time and hybrid systems using QSS algorithms. Also, we described their implementation in a software simulation tool.

The methodologies are based on the use of non strict synchronization between logical processes and introduce an additional numerical error to that of the QSS approximation. We showed that this additional error is bounded depending on a parameter Δt, and we designed an adaptive algorithm for dynamically

Table 12
Spiking neurons model — fixed $\Delta t = 5e-1$.

Cores	Init time (ms)	Simulation time (s)	Memory (MBytes)	Speedup	Average spikes
1	1244	5402	2,555	1.00	15.84
2	1806	3331	2,916	1.62	15.86
4	1809	2030	3,670	2.66	15.94
8	2071	1349	5,177	4.00	15.85
16	2176	1012	8,257	5.33	15.93
32	2594	439	14,352	12.30	16.16
48	3104	288	20,480	18.75	15.81
62	3536	231	25,851	23.38	15.80

Table 14
Spiking neurons model — adaptive $\Delta t - \alpha = 10$.

Cores	Init time (ms)	Simulation time (s)	Memory (MBytes)	Speedup	Average spikes
1	1244	5402	2,555	1.00	15.84
2	1805	3367	2,916	1.60	16.41
4	1806	2040	3,670	2.64	15.50
8	1935	1358	5,177	3.97	16.04
16	2196	1049	8,257	5.14	15.79
32	2654	486	14,352	11.11	15.77
48	3123	354	20,480	15.25	15.93
62	3512	344	25,851	15.70	15.84

computing that parameter in order to keep the error bounded according to the desired tolerance.

The results obtained on four large systems (one of them being purely continuous and the remaining having discontinuities) showed linear speedups with the number of cores. For a maximum of 62 cores used, the speedup reached values between 22 and 45 in the different cases. Taking into account that the sequential QSS simulations of those examples are faster than those of classic discrete time numerical algorithms, the parallel QSS results can increase the advantages.

This is the first work reporting parallel simulations with QSS methods in models with a size of the order of one million variables (accounting states and discontinuity handlers).

Regarding future work, we consider the following items:

- The results should be extended to a wider set of models, including in particular other types of PDEs.
- Although the tool includes the automatic partitioning algorithms of [12], they must be adapted and optimized according to the features of this particular parallel implementation.
- In many problems, like in that of the Advection–Reaction equation, the results can be improved using dynamic load balancing. To this end, we plan to extend the tool implementing dynamic partitioning algorithms.
- In order to simulate using more LPs, we need to extend the techniques and the implementation to consider distributed memory architectures, where the synchronization strategy must be modified.
- We are currently working on the automatic generation of partitioned sub-models in order to facilitate the use of the model partitioned implementation, which is far more efficient in terms of memory consumption.

The Parallel Stand-Alone QSS solver is an open source project, available at the site http://sourceforge.net/projects/qssengine. The models used in this article are part of the distribution.

Acknowledgment

This work was partially funded with grant ANPCYT PICT 2012-0077.

References

[1] P. Amodio, L. Brugnano, Parallel solution in time of odes: some achievements and perspectives, Appl. Numer. Math. 59 (3) (2009) 424–435.
[2] R. Assar, D.J. Sherman, Implementing biological hybrid systems: Allowing composition and avoiding stiffness, Appl. Math. Comput. 223 (2013) 167–179.
[3] P.D. Barnes Jr., C.D. Carothers, D.R. Jefferson, J.M. LaPre, Warp speed: executing time warp on 1,966,080 cores, in: Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, ACM, 2013, pp. 327–336.
[4] F. Bergero, J. Fernández, E. Kofman, M. Portapila, Time discretization versus state quantization in the simulation of a 1D advection–diffusion–reaction equation, Simulation 92 (1) (2016) 47–61.
[5] F. Bergero, E. Kofman, PowerDEVS. A tool for hybrid system modeling and real time simulation, Simulation 87 (1–2) (2011) 113–132.
[6] F. Bergero, E. Kofman, F.E. Cellier, A novel parallelization technique for DEVS simulation of continuous and hybrid systems, Simulation 89 (6) (2013) 663–683.
[7] G.D. Byrne, A.C. Hindmarsh, Pvode, an ode solver for parallel computers, Int. J. High Perform. Comput. Appl. 13 (4) (1999) 354–365.
[8] F. Cellier, E. Kofman, Continuous System Simulation, Springer, New York, 2006.
[9] S. Chatzivasileiadis, M. Bonvini, J. Matanza, R. Yin, T.S. Nouidui, E.C. Kara, R. Parmar, D. Lorenzetti, M. Wetter, S. Kiliccote, Cyber-physical modeling of distributed resources for distribution system operations, Proc. IEEE 104 (4) (2016) 789–806.
[10] C. Farhat, M. Chandesris, Time-decomposed parallel time-integrators: Theory and feasibility studies for uid, structure, and fluid–structure applications, Internat. J. Numer. Methods Engrg. 58 (2003) 1397–1434.
[11] J. Fernández, E. Kofman, A stand-alone quantized state system solver for continuous system simulation, Simulation 90 (7) (2014) 782–799.
[12] X. Floros, Exploiting model structure for efficient hybrid dynamical systems simulation (Ph.D. thesis, Diss.), Eidgenössische Technische Hochschule ETH, Zürich, 2014, Nr. 21871, 2014.
[13] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley-Interscience, New York, 2004.
[14] R.M. Fujimoto, Parallel and Distributed Simulation Systems, Vol. 300, Wiley, New York, 2000.
[15] R. Fujimoto, Parallel and distributed simulation, in: Proceedings of the 2015 Winter Simulation Conference, IEEE Press, 2015, pp. 45–59.
[16] S. Ghoshal, M. Gupta, V. Rajaraman, A parallel multistep predictor–corrector algorithm for solving ordinary differential equations, J. Parallel Distrib. Comput. 6 (3) (1989) 636–648.
[17] G. Grinblat, H. Ahumada, E. Kofman, Quantized state simulation of spiking neural networks, Simulation 88 (3) (2012) 299–313.
[18] M. Rodríguez, F. Blesa, R. Barrio, Opencl parallel integration of ordinary differential equations: Applications in computational dynamics, Comput. Phys. Comm. 192 (2015) 228–236.
[19] E. Hairer, S. Norsett, G. Wanner, Solving Ordinary Differential Equations I. Nonstiff Problems, second ed., Springer, 1993.
[20] F. Iavernaro, F. Mazzia, Generalization of backward differentiation formulas for parallel computers, Numer. Algorithms 31 (1–4) (2002) 139–155.
[21] S. Jafer, G. Wainer, Global lookahead management (glm) protocol for conservative devs simulation, in: 2010 IEEE/ACM 14th International Symposium on Distributed Simulation and Real Time Applications (DS-RT), IEEE, 2010, pp. 141–148.
[22] D.R. Jefferson, Virtual time, ACM Trans. Program. Lang. Syst. 7 (3) (1985) 404–425.
[23] N. Kalinnik, M. Korch, T. Rauber, Online auto-tuning for the time-step-based parallel solution of odes on shared-memory systems, J. Parallel Distrib. Comput. 74 (8) (2014) 2722–2744.
[24] E. Kofman, A second order approximation for DEVS simulation of continuous systems, Simulation 78 (2) (2002) 6–89.
[25] E. Kofman, Discrete event simulation of hybrid systems, SIAM J. Sci. Comput. 25 (5) (2004) 1771–1797.
[26] E. Kofman, A third order discrete event simulation method for continuous system simulation, Lat. Am. Appl. Res. 36 (2) (2006) 101–108.
[27] E. Kofman, Relative error control in quantization based integration, Lat. Am. Appl. Res. 39 (3) (2009) 231–238.
[28] E. Kofman, S. Junco, Quantized state systems. A DEVS approach for continuous system simulation, Trans. SCS 18 (3) (2001) 123–132.
[29] M. Korch, T. Rauber, Optimizing locality and scalability of embedded Runge–Kutta solvers using block-based pipelining, J. Parallel Distrib. Comput. 66 (3) (2006) 444–468.
[30] M. Korch, T. Rauber, Locality optimized shared-memory implementations of iterated runge-kutta methods, in: Euro-Par 2007 Parallel Processing, Springer, 2007, pp. 737–747.
[31] J. Lions, Y. Maday, G. Turinici, A "parareal" in time discretization of pde's, C. R. Acad. Sci., Paris I 332 (7) (2001) 661–668.
[32] M. Maggio, K. Stavåker, F. Donida, F. Casella, P. Fritzson, Parallel simulation of equation-based object-oriented models with quantized state systems on a GPU, in: Proceedings of the 7th International Modelica Conference, 2009.
[33] G. Migoni, M. Bortolotto, E. Kofman, F. Cellier, Linearly implicit quantization-based integration methods for stiff ordinary differential equations, Simul. Model. Pract. Theory 35 (2013) 118–136.

[34] L. Mikelsons, N. Menager, D. Schramm, Partitioned model vs parallelized solver, in: ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, American Society of Mechanical Engineers, 2011, pp. 1101–1109.
[35] J. Misra, Distributed discrete-event simulation, ACM Comput. Surv. 18 (1) (1986) 39–65.
[36] L. Murray, Gpu acceleration of Runge–Kutta integrators, IEEE Trans. Parallel Distrib. Syst. 23 (1) (2012) 94–101.
[37] J.J. Nutaro, Parallel discrete event simulation with application to continuous systems (Ph.D. thesis), University of Arizona, 2003.
[38] D. Paluszczyszyn, P. Skworcow, B. Ulanicki, Modelling and simulation of water distribution systems with quantised state system methods, Procedia Eng. 119 (2015) 554–563.
[39] C. Perfumo, E. Kofman, J. Braslavsky, J. Ward, Load management: Model-based control of aggregate power for populations of thermostatically controlled loads, Energy Convers. Manage. 55 (2012) 36–48.
[40] D. Petcu, Experiments with an ode solver on a multiprocessor system, Comput. Math. Appl. 42 (8) (2001) 1189–1199.
[41] D.M. Rao, Efficient parallel simulation of spatially-explicit agent-based epidemiological models, J. Parallel Distrib. Comput. 93 (2016) 102–119.
[42] D. Rao, N. Thondugulam, R. Radhakrishnan, P. Wilsey, Unsynchronized parallel discrete event simulation, in: Simulation Conference Proceedings, 1998. Winter, Vol. 2, 1998, pp. 1563–1570.
[43] L. Santi, N. Ponieman, K. Genser, V.D. Elvira, Y. Jun Soon, R. Castro, Application of state quantization-based methods in hep particle transport simulation, in: 22nd International Conference on Computing in High Energy and Nuclear Physics, CHEP 2016, San Francisco, CA, 2016.
[44] V.M. Soto Frances, E.J. Sarabia Escriva, J.M. Pinazo Ojer, Discrete event heat transfer simulation of a room, Int. J. Therm. Sci. 97 (2015) 82–93.
[45] T.P. Vogels, L.F. Abbott, Signal propagation and logic gating in networks of integrate-and-fire neurons, J. Nuerosci. 25 (46) (2005) 10786–10795.
[46] G.A. Wainer, The cell-devs formalism as a method for activity tracking in spatial modelling and simulation, Int. J. Simul. Process Modelling 10 (1) (2015) 19–38.
[47] M. Wetter, T.S. Nouidui, D. Lorenzetti, E.A. Lee, A. Roth, Prototyping the next generation energyplus simulation engine, in: 13th IBPSA Conference, International Building Performance Simulation Association, 2015.

Joaquín Fernández received his B.S. degree in Computer Science in 2012 from the Universidad Nacional de Rosario, Argentina. He is currently a Ph.D. student at the French Argentine International Center for Information and Systems Sciences (CIFASIS). His research interests include hybrid system simulation, real-time and parallel simulation, simulation tools, and modeling languages.

Ernesto Kofman received his B.S. degree in electronic engineering in 1999 and his Ph.D. degree in automatic control in 2003, all from the National University of Rosario. He is an adjunct professor at the department of control of the National University of Rosario (FCEIA–UNR). He also holds a research position at the National Research Council of Argentina (CIFASIS-CONICET). His research interests include automatic control theory and numerical simulation of hybrid systems. He coauthored a textbook on continuous system simulation in 2006 with Springer, New York.

Federico Bergero received a computer science degree in 2008 from the Universidad Nacional de Rosario, Argentina. He is currently a Ph.D. student at the French Argentine International Center for Information and Systems Sciences (CIFASIS) and he holds a teaching position at the Universidad Nacional de Rosario in the department of computer science, FCEIA-UNR, Rosario, Argentina. His research interests include discrete event systems, real time and parallel simulation, signal processing, and audio filtering methods.